Efficient zero-copy IO

and

Immutable memory management

in

Midori

Jinsong Yu

May 2013

Microsoft

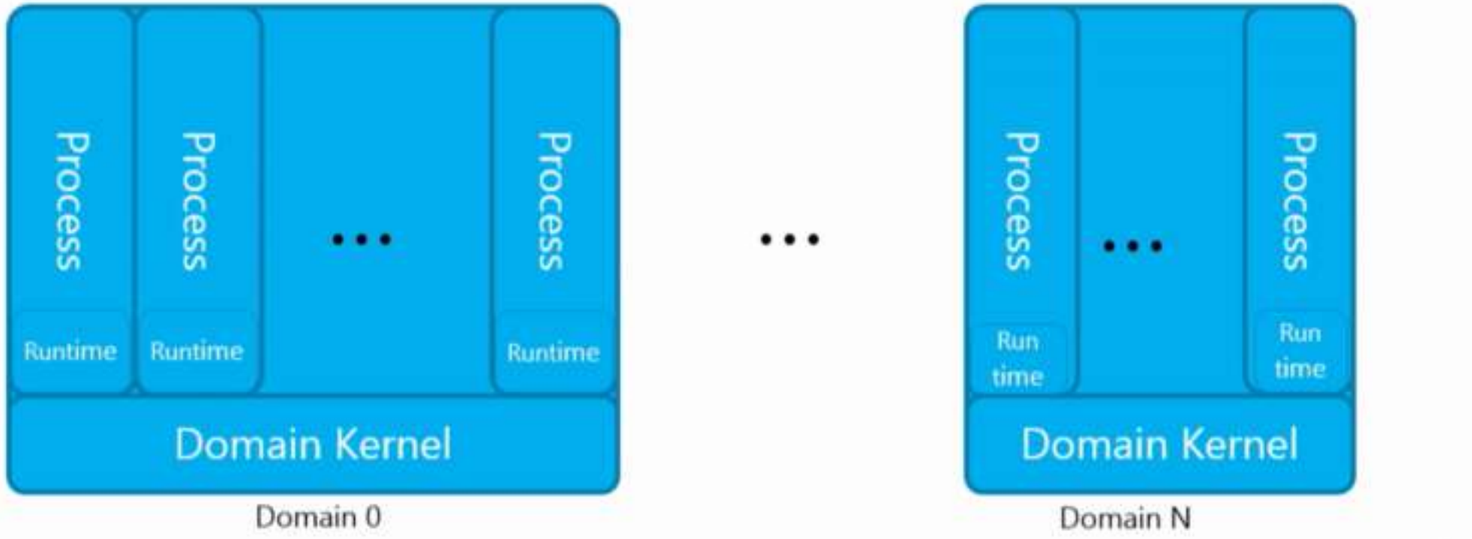
Microkernel Architecture

- Each device drivers and IO service is a user mode process
- No IO through kernel

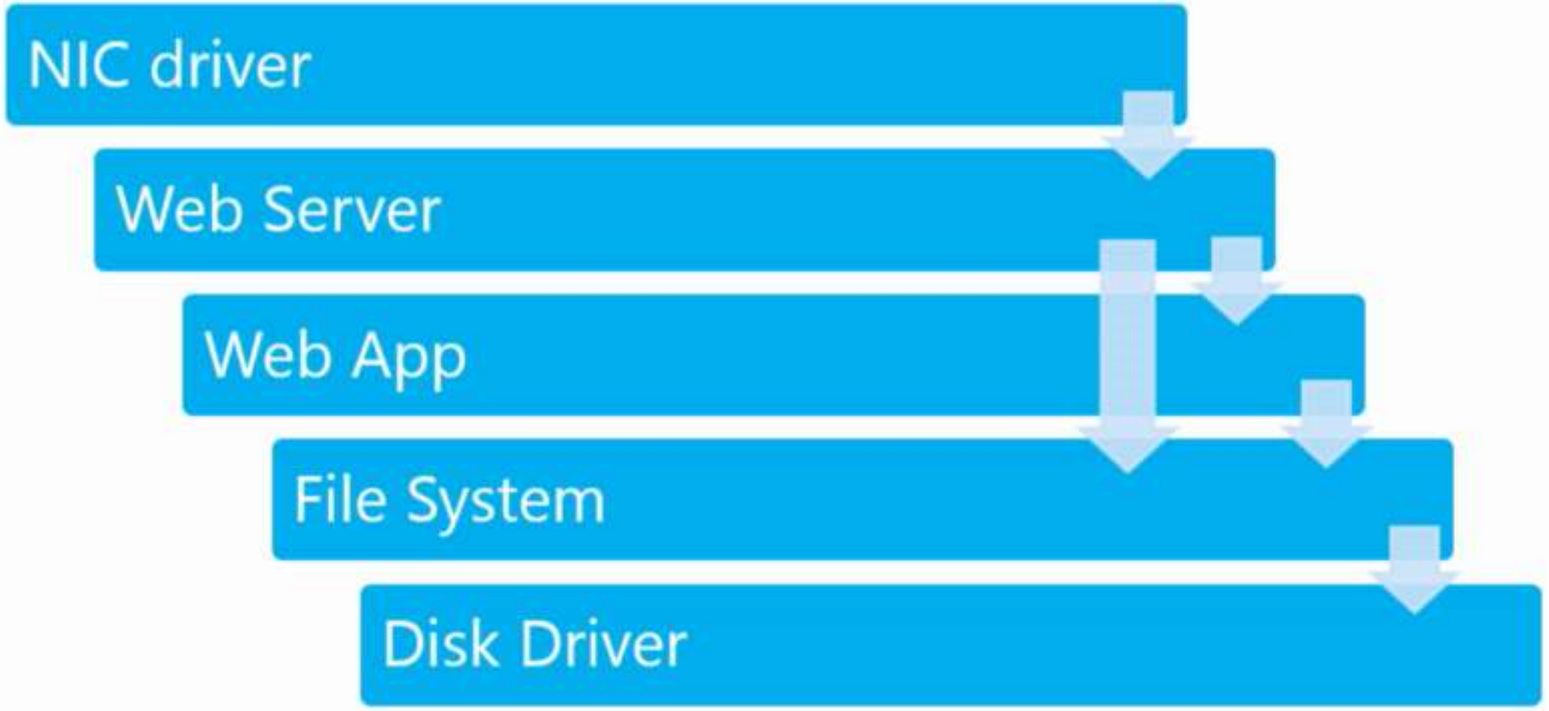
Software Isolated Processes

- Multiple processes live in the same hardware address space called domains
- Processes are strongly isolated from each other using memory/type safety
- Each process has its own GC heap

System Architecture



Example: Web App IO Pipeline



- With data travelling through many processes, we need an efficient data transfer mechanism.
- Data ***must not*** live on the GC heap

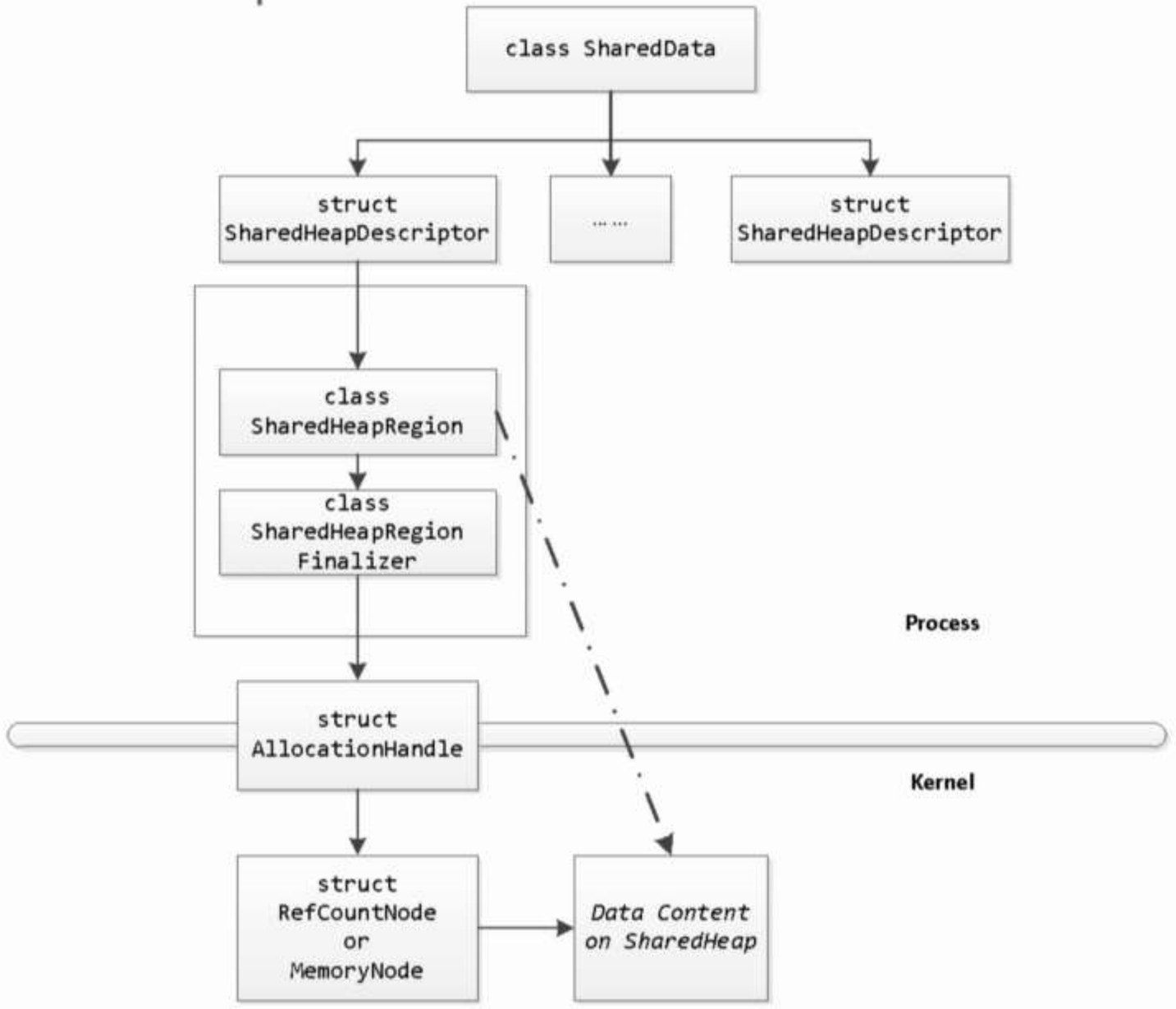
Shared Heap

- Support bulk data sharing and transfer in IO path
- One SharedHeap per domain
- No relocation
- No typed objects, pointers, or references
- Process level lifetime management and accounting
- Data can be
  - exclusive: single owner, content is mutable
  - or shared: potentially multiple owners, content is immutable
- Lifetime is managed by handles, exposed to app code as Stream, SharedData, or SharedMultiSpan

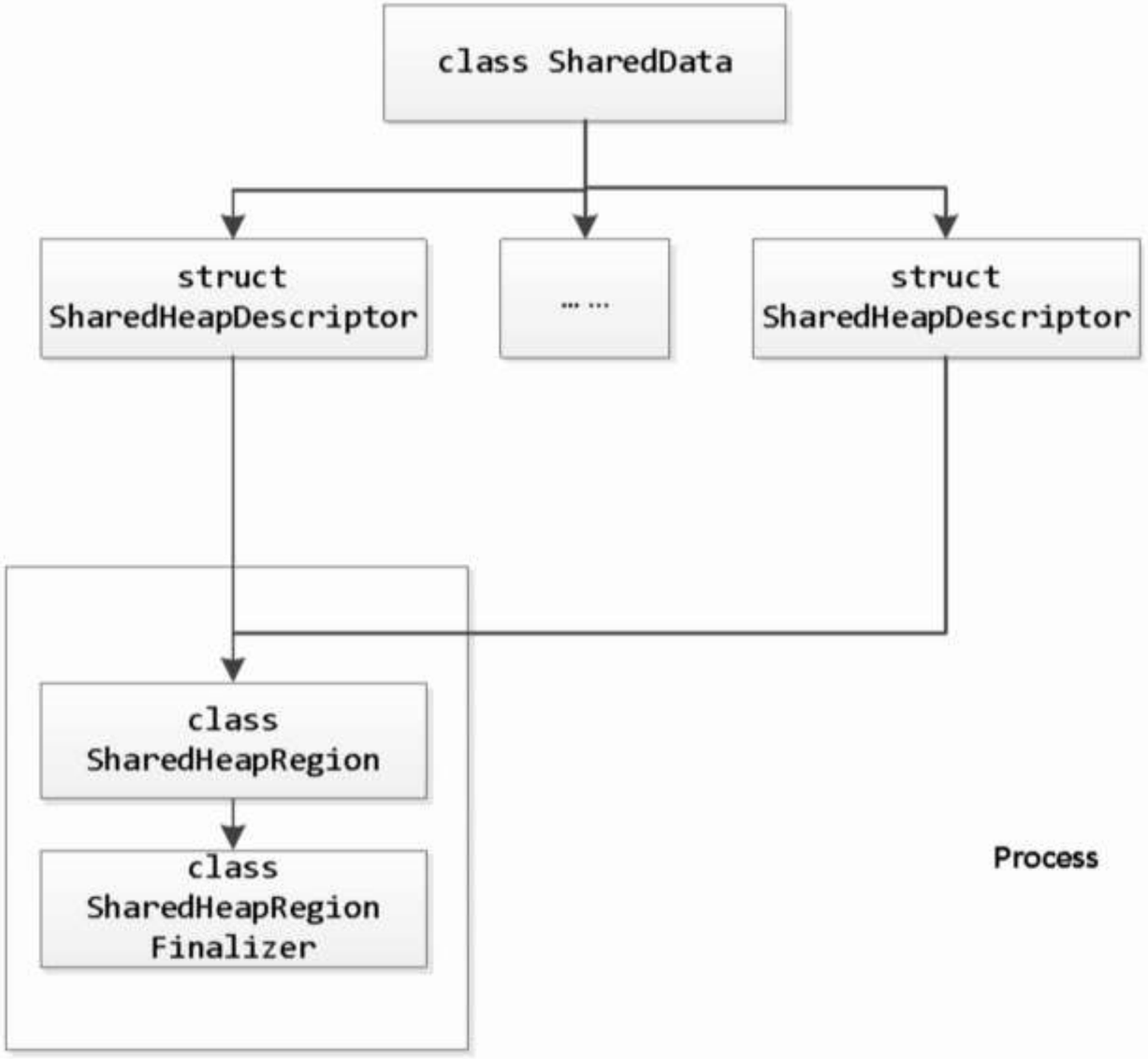
SharedData

- Small GC Heap object to represent the data on the Shared Heap
- Manages data life time, guarantee no use-after-free
- Can have multiple segments
- Zero-copy to concatenate, append headers, strip off headers
- IDisposable and Finalizable
- Content is immutable
- Can be transferred or shared across process boundary efficiently

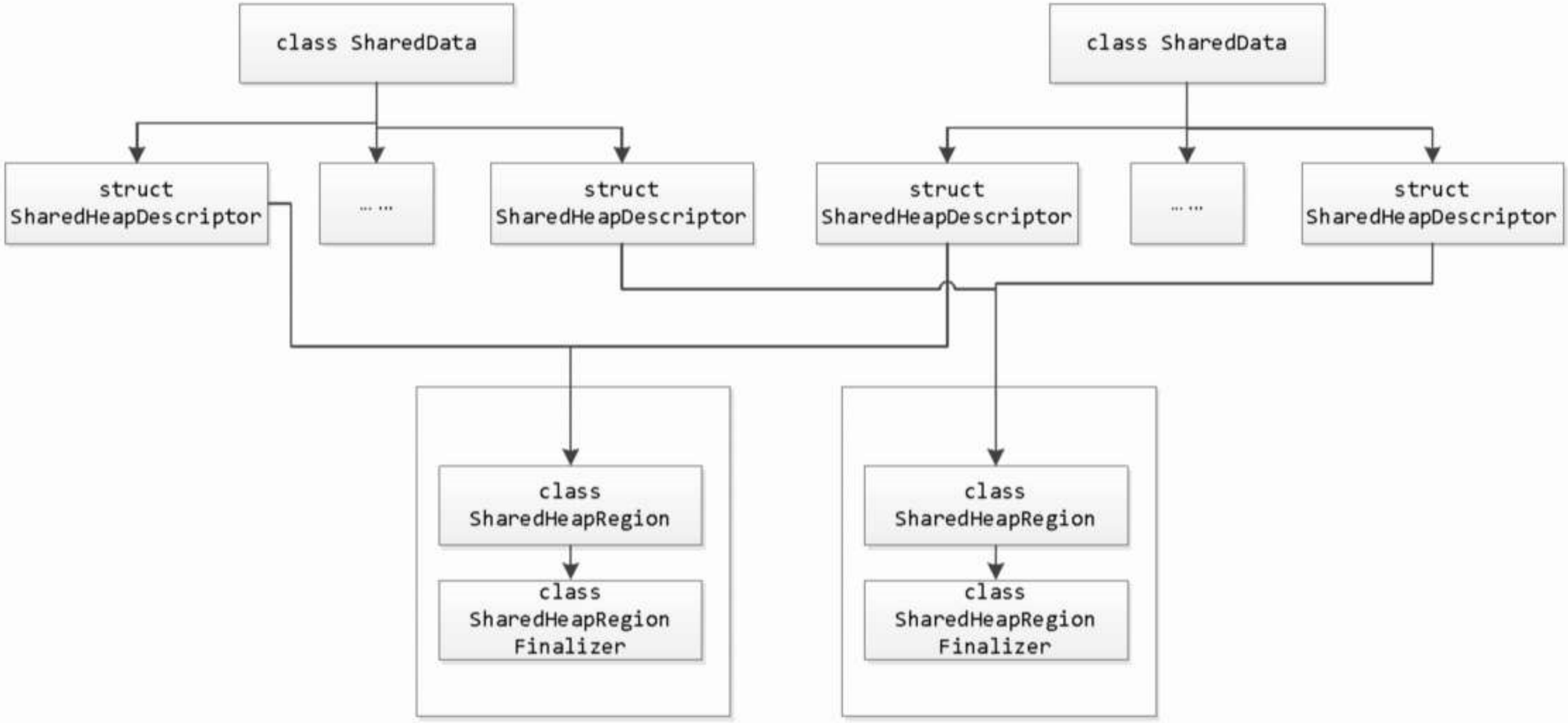
SharedData Implementation



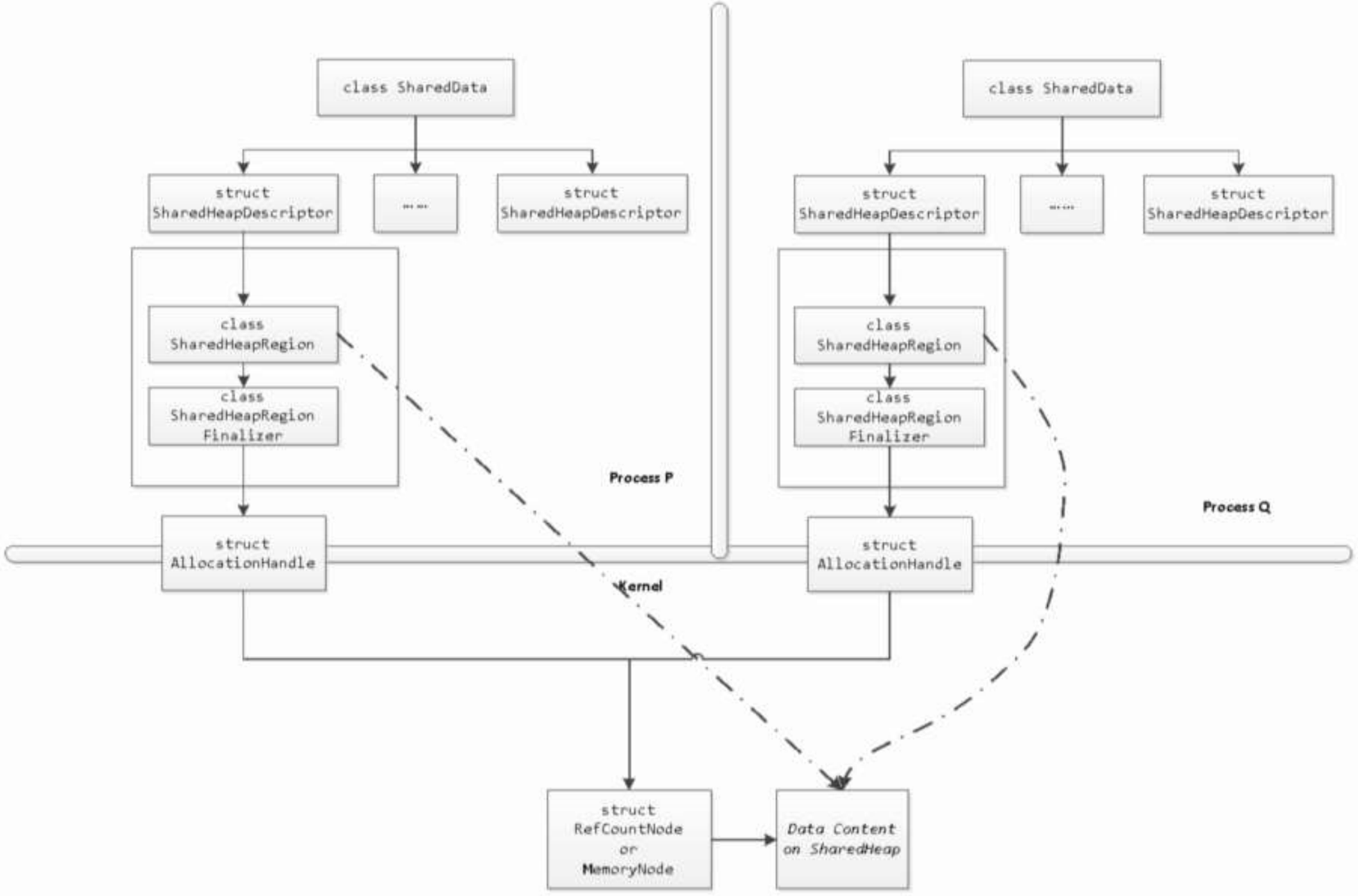
Example of Sharing #1



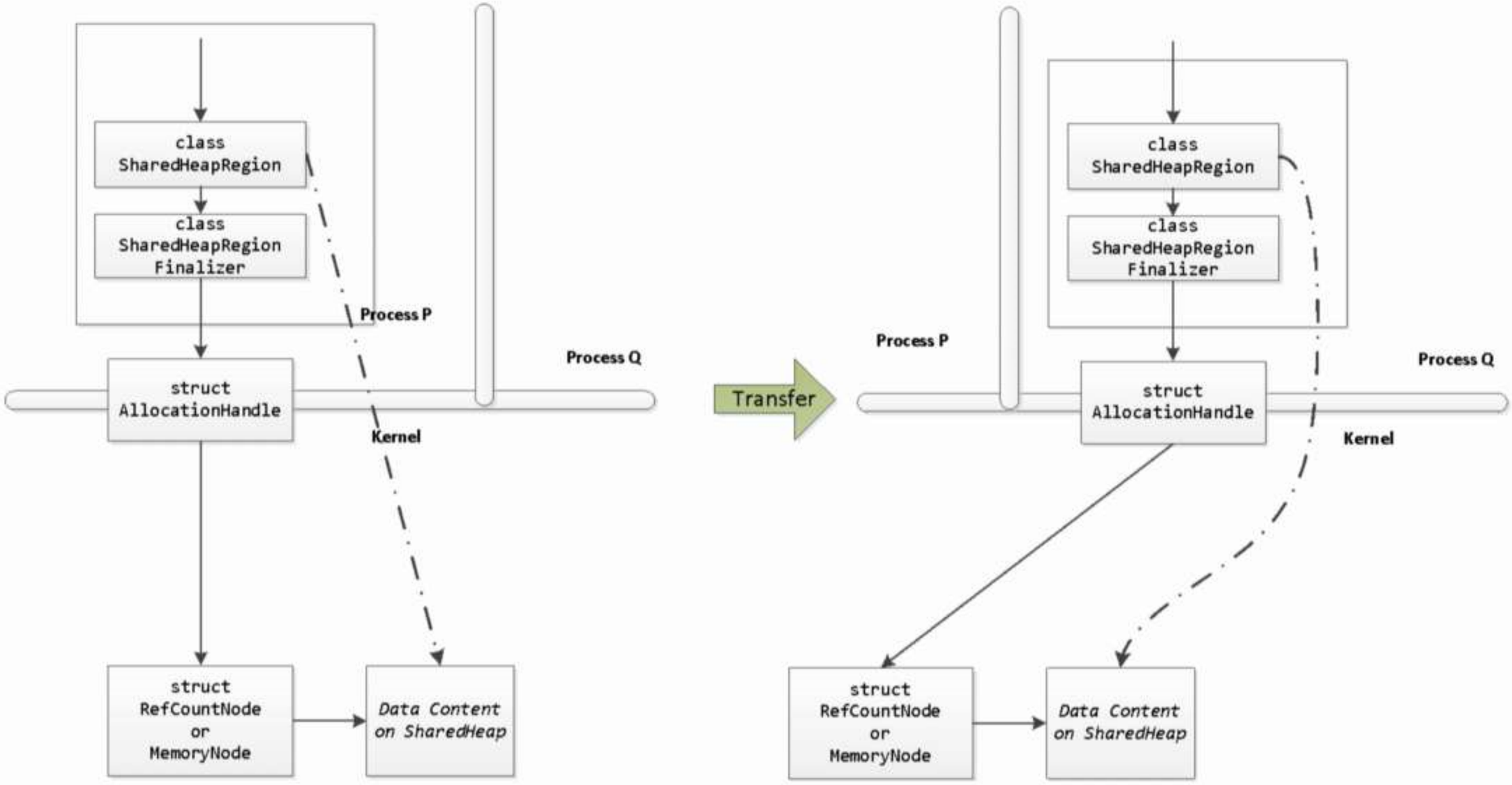
Example of Sharing #2



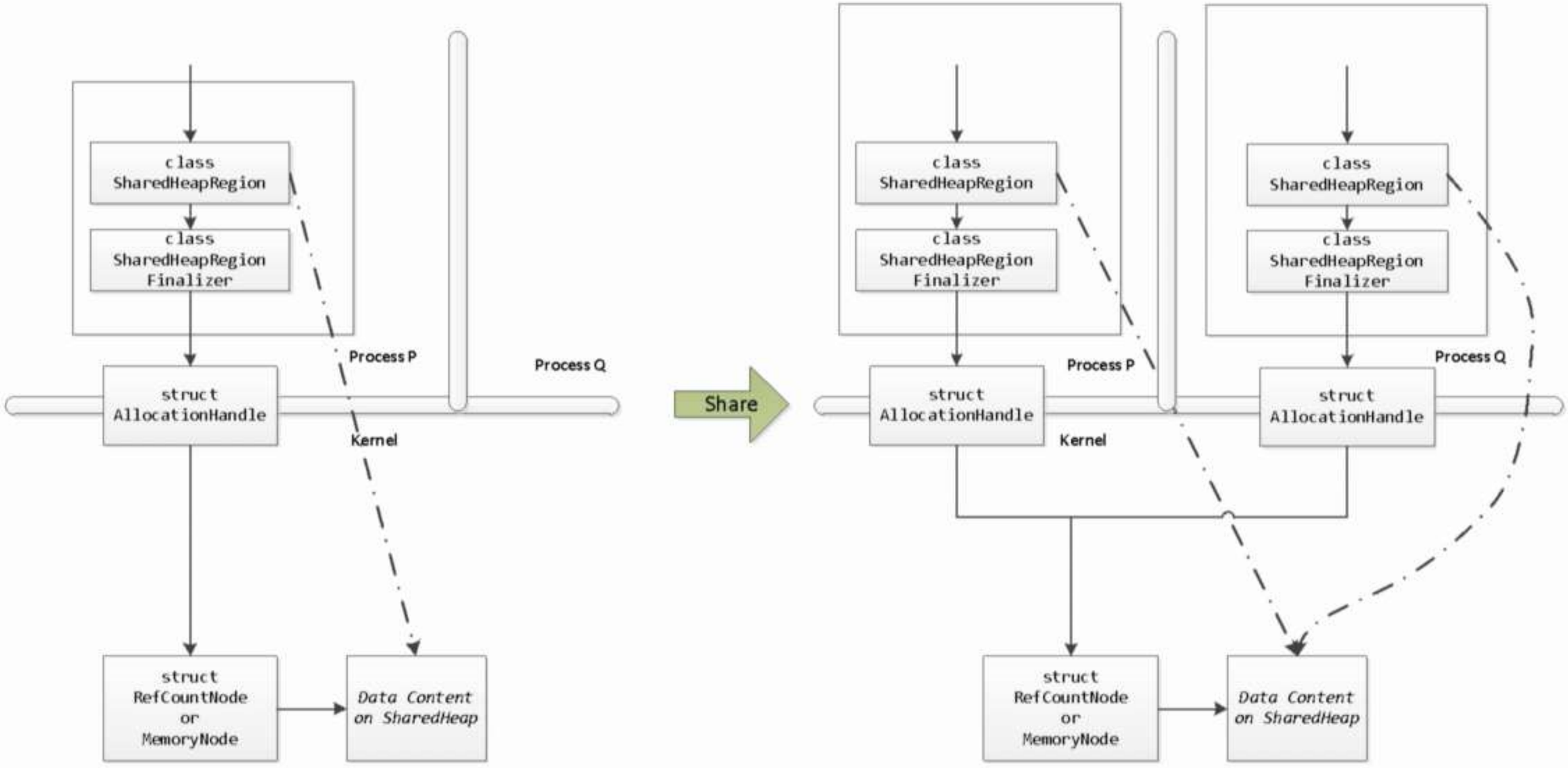
Example of Sharing #3



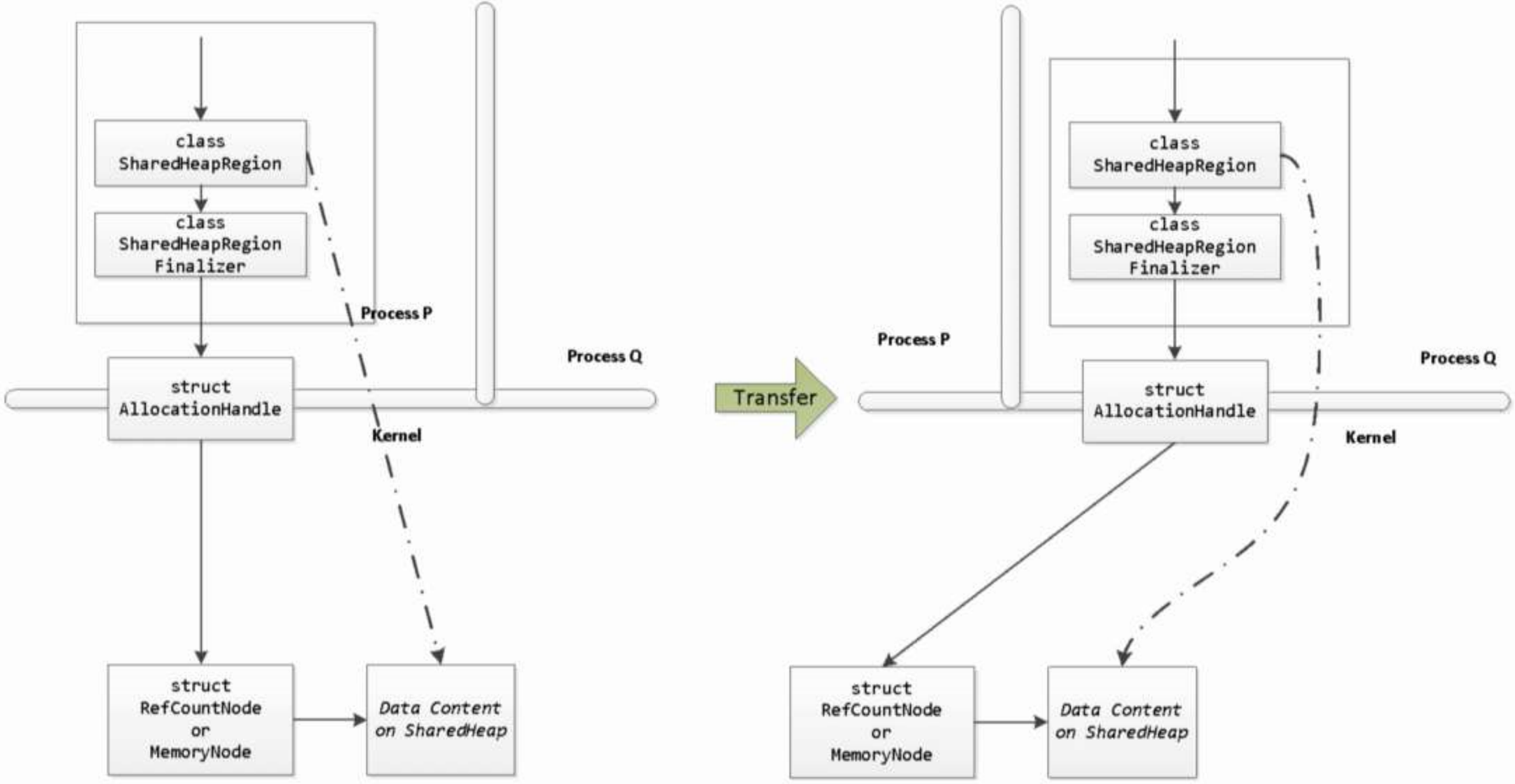
SharedData Transfer



SharedData Share



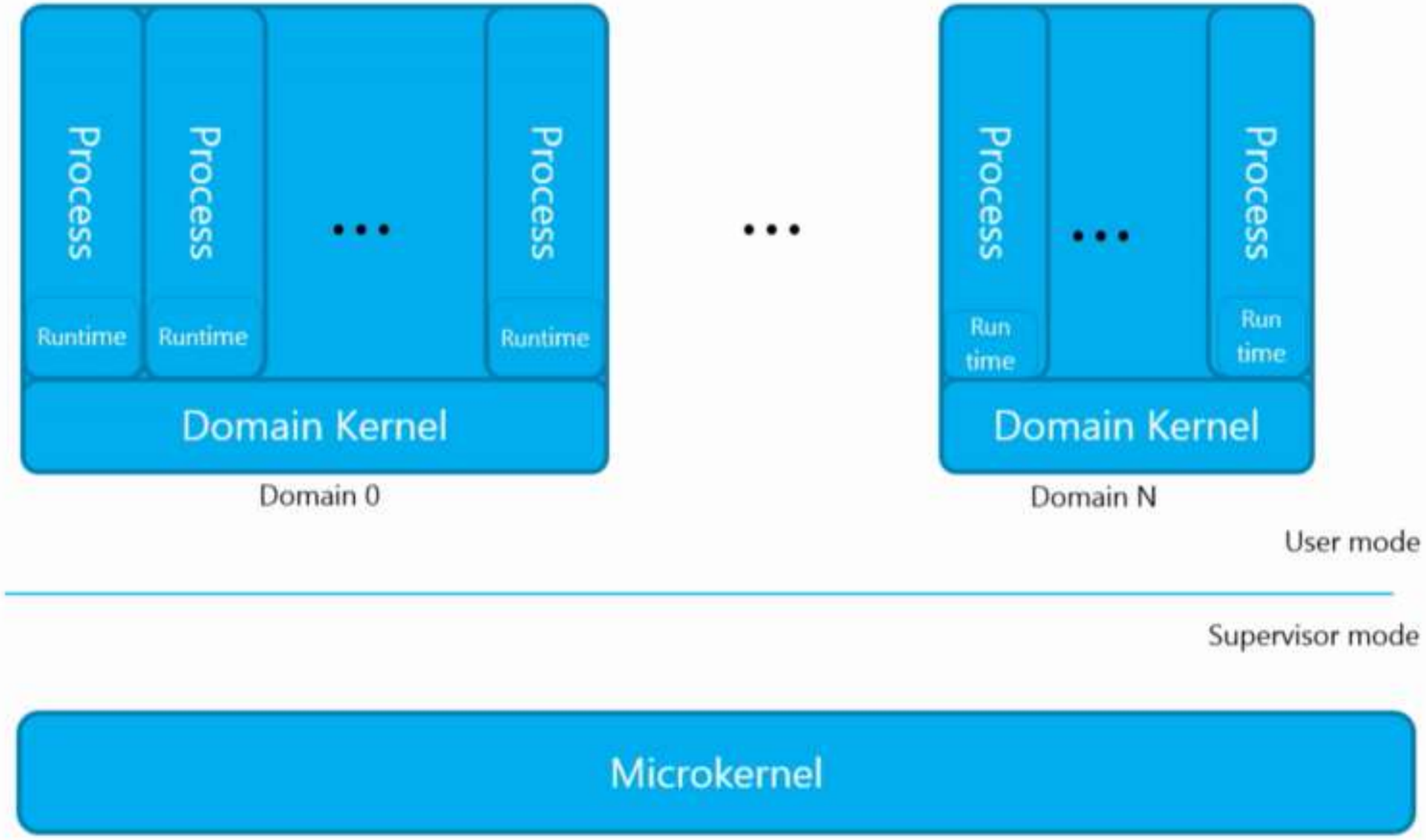
SharedData Transfer



Only Immutable Data Can Be Shared

- Avoid data races, TOCTOU
- Minimize CPU cache coherency protocol overhead
- Allows simple and efficient caching
- Seems limiting, but this restriction turns out to be a great ally in system performance

System Architecture



Unifying Data Access: Span<T>

- APIs using Span<byte> can be used to process data on both GC Heap and Shared Heap.
- Deep runtime and class library integration

```
byte[] gcData = ...;
Span<byte> s1 = gcData.GetSpan();
long decodedCharCount1 = Utf8Encoding.GetCharCount(in s1);

SharedData shData = ...;
Span<byte> s2 = shData.GetLongestSpanAt(0);
long decodedCharCount2 = Utf8Encoding.GetCharCount(in s2);
```

Primitive Structs

- Structs composed of only primitive types, no references
- APIs to read/write/cast primitive structs

```
// Both T and U below are primitive struct types
Span<byte> a = ...;
T t = a.Read<T>(offset);
a.Write<T>(offset, t);

Span<T> b = ...;
Span<U> c = Primitive.Cast<T, U>(b);
```

Example

Reading ICMP header from a network packet

```
public primitive struct IcmpHeader : IInteroperable
{
    public IcmpType Type;
    public byte Code;
    public BigEndian.UInt16 Checksum;
    public BigEndian.UInt16 Identifier;
    public BigEndian.UInt16 Sequence;
    ...
}

SharedData data = ... // The network packet
IcmpHeader h = data.Read<IcmpHeader>(offset);
```

Primitive Structs

- Structs composed of only primitive types, no references
- APIs to read/write/cast primitive structs

```
// Both T and U below are primitive struct types
Span<byte> a = ...;
T t = a.Read<T>(offset);
a.Write<T>(offset, t);

Span<T> b = ...;
Span<U> c = Primitive.Cast<T, U>(b);
```

Example

Reading ICMP header from a network packet

```
public primitive struct IcmpHeader : IInteroperable
{
    public IcmpType Type;
    public byte Code;
    public BigEndian.UInt16 Checksum;
    public BigEndian.UInt16 Identifier;
    public BigEndian.UInt16 Sequence;
    ...
}

SharedData data = ... // The network packet
IcmpHeader h = data.Read<IcmpHeader>(offset);
```

Destructible Resource, SharedMultiSpan

- Allocated on the stack, not GC Heap
- C++ destructor semantics
- Enforced by language and compiler
- No IDisposable, no finalization
- SharedMultiSpan is otherwise similar to SharedData

Unifying Data Access: Span<T>

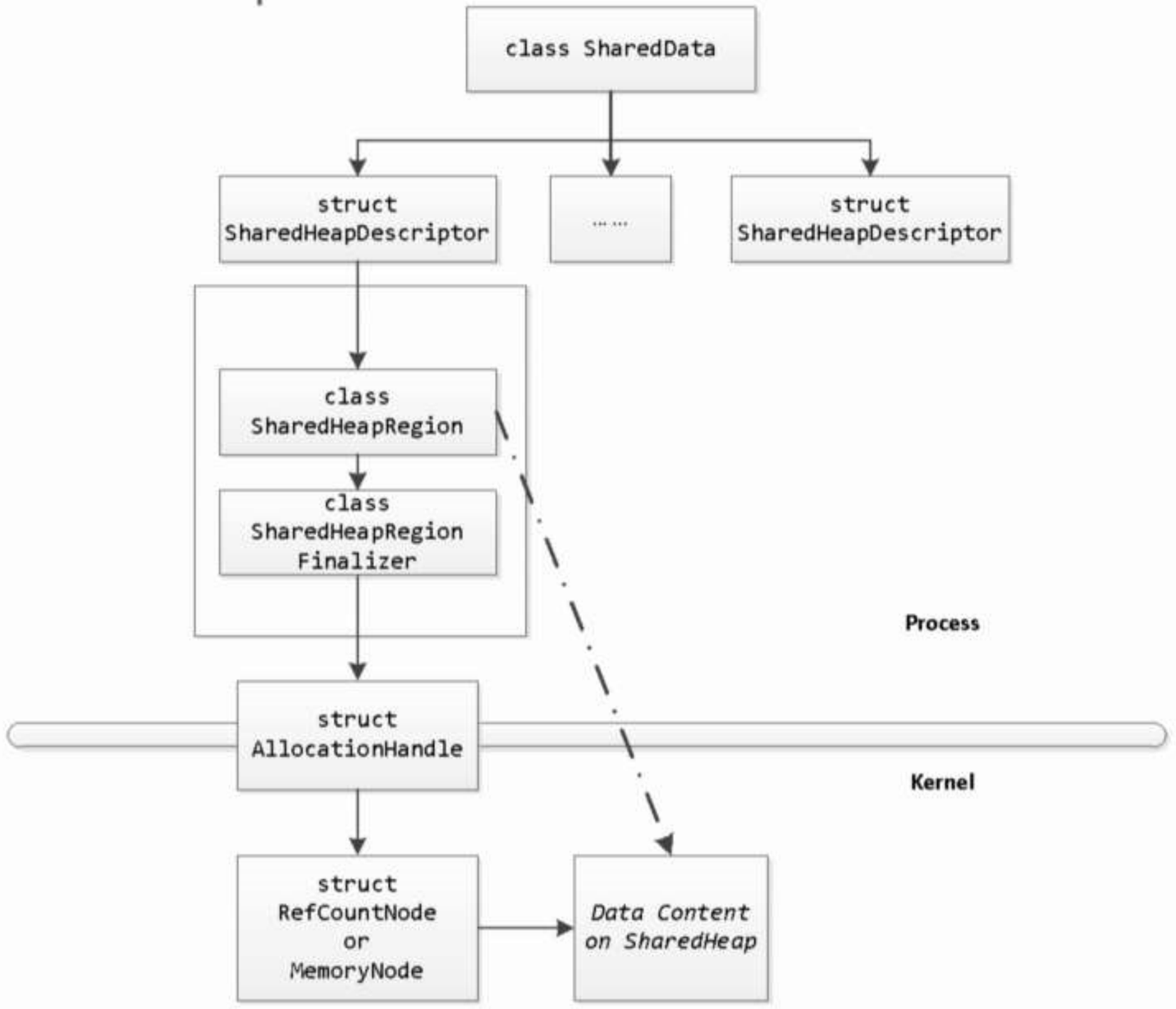
- APIs using Span<byte> can be used to process data on both GC Heap and Shared Heap.
- Deep runtime and class library integration

```
byte[] gcData = ...;
Span<byte> s1 = gcData.GetSpan();
long decodedCharCount1 = Utf8Encoding.GetCharCount(in s1);

SharedData shData = ...;
Span<byte> s2 = shData.GetLongestSpanAt(0);
long decodedCharCount2 = Utf8Encoding.GetCharCount(in s2);
```

Destructible Resource, SharedMultiSpan

- Allocated on the stack, not GC Heap
- C++ destructor semantics
- Enforced by language and compiler
- No IDisposable, no finalization
- SharedMultiSpan is otherwise similar to SharedData

SharedData Implementation



Example

Reading ICMP header from a network packet

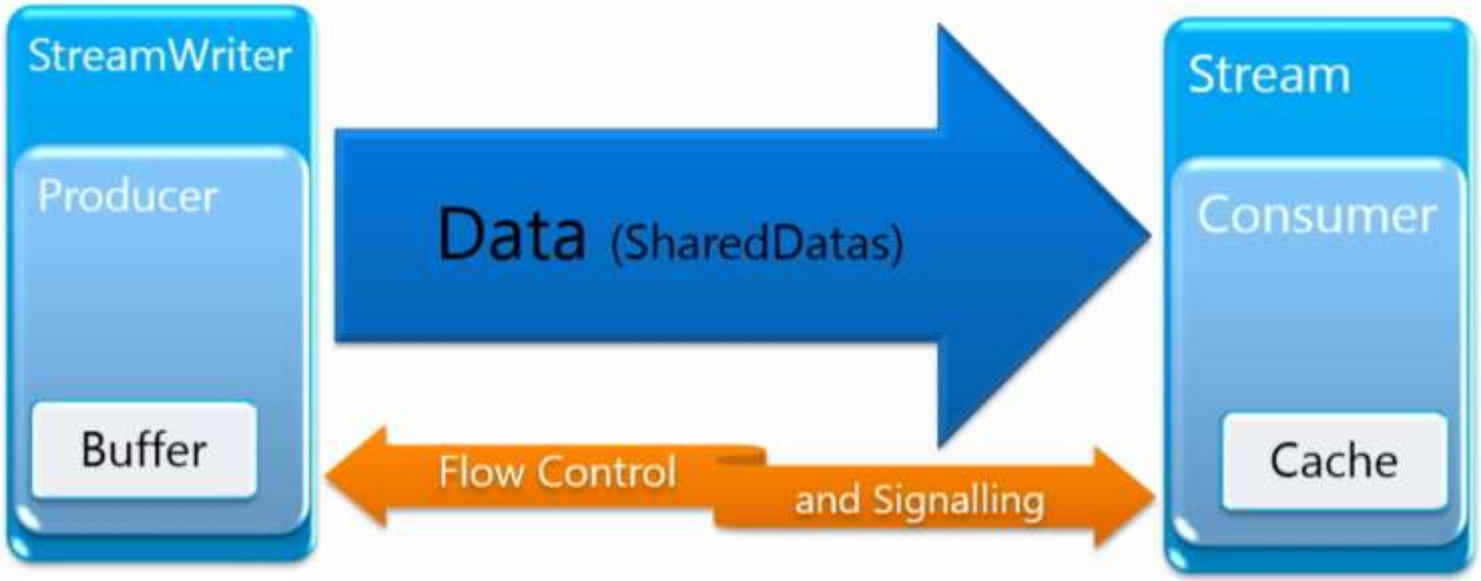
```
public primitive struct IcmpHeader : IInteroperable
{
    public IcmpType Type;
    public byte Code;
    public BigEndian.UInt16 Checksum;
    public BigEndian.UInt16 Identifier;
    public BigEndian.UInt16 Sequence;
    ...
}

SharedData data = ... // The network packet
IcmpHeader h = data.Read<IcmpHeader>(offset);
```

Destructible Resource, SharedMultiSpan

- Allocated on the stack, not GC Heap
- C++ destructor semantics
- Enforced by language and compiler
- No IDisposable, no finalization
- SharedMultiSpan is otherwise similar to SharedData

Stream

- A potentially large sequence of bytes containing unstructured data
- Producer and consumer may exist in different processes
- May be long lived
- Important mechanism to marshal data



SharedData Soft/Weak Reference

- Used primarily in caches
- Have no methods to access data. To access data, must convert to SharedData first
- Lifetime rules:
  - If there are SharedData referencing a data region, the data region is guaranteed to remain alive
  - If there are neither SharedData nor SharedDataSoftReferences referencing a data region, regardless whether there are SharedDataWeakReferences referencing the data region, it's guaranteed to be freed
  - If there are no SharedData referencing a data region, but there are SharedDataSoftReferences referencing it, regardless whether there are SharedDataWeakReferences, the data region is in average case kept alive. Only if kernel run out of memory and choose to steal memory from that data region, it is freed.

SharedDataCache

- Most data are in SharedDataSoftReference
- Small subset of data are in SharedData
- Used by backend, large caches (e.g. storage stack)
- Allows the system to use most physical memory for caching purpose without demand paging

SharedData Soft/Weak Reference

- Used primarily in caches
- Have no methods to access data. To access data, must convert to SharedData first
- Lifetime rules:
  - If there are SharedData referencing a data region, the data region is guaranteed to remain alive
  - If there are neither SharedData nor SharedDataSoftReferences referencing a data region, regardless whether there are SharedDataWeakReferences referencing the data region, it's guaranteed to be freed
  - If there are no SharedData referencing a data region, but there are SharedDataSoftReferences referencing it, regardless whether there are SharedDataWeakReferences, the data region is in average case kept alive. Only if kernel run out of memory and choose to steal memory from that data region, it is freed.

SharedDataCache

- Most data are in SharedDataSoftReference
- Small subset of data are in SharedData
- Used by backend, large caches (e.g. storage stack)
- Allows the system to use most physical memory for caching purpose without demand paging

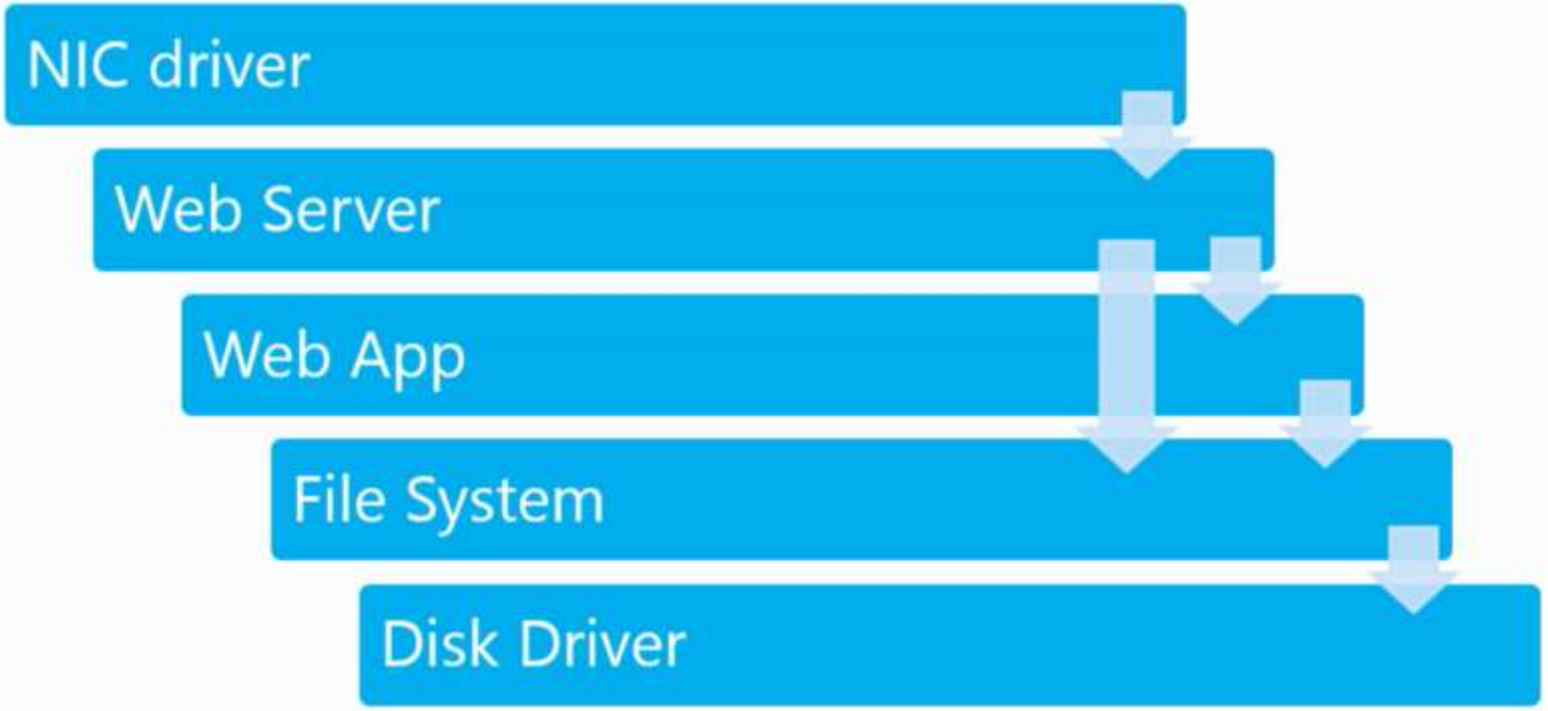
SharedDataWeakReferenceCache

- Only hold SharedDataWeakReference
- Work as front end cache
  - Reduce IPC overhead to communicate with backend cache
  - Reduce load on backend cache

Example:

- Web server static content cache

Example: Web App IO Pipeline



- With data travelling through many processes, we need an efficient data transfer mechanism.
- Data ***must not*** live on the GC heap

SharedDataWeakReferenceCache

- Only hold SharedDataWeakReference
- Work as front end cache
  - Reduce IPC overhead to communicate with backend cache
  - Reduce load on backend cache

Example:

- Web server static content cache

Result: Speech Server Performance

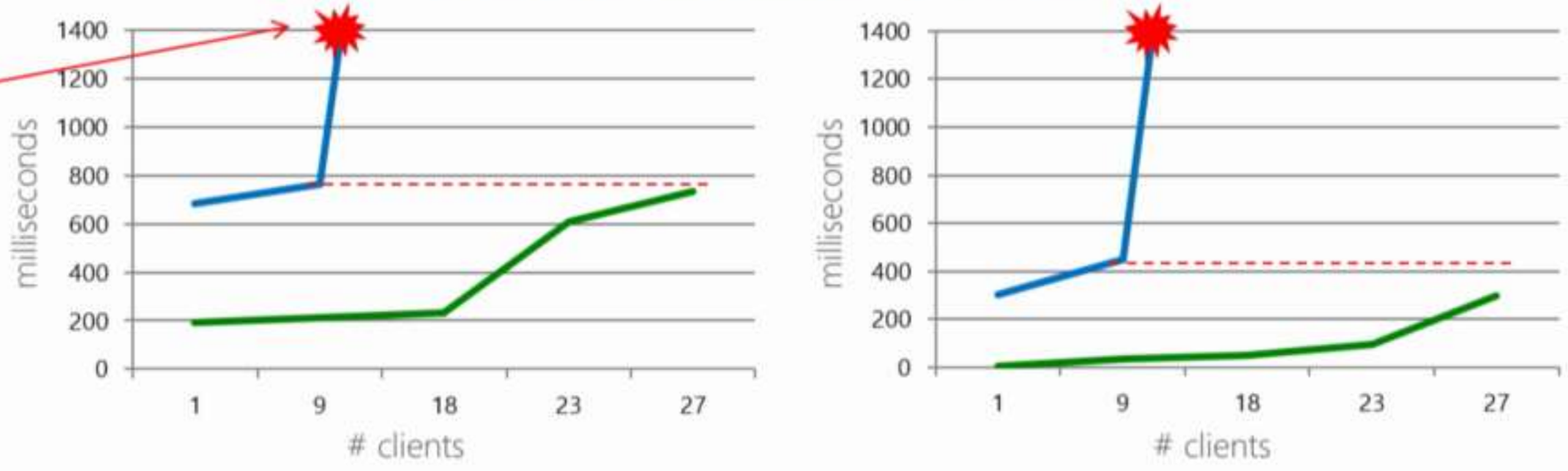
At more than triple the load, Midori delivers customer-visible latency gains

- Windows Phone beats latency target at 3x load; Xbox at 4x

	Windows Phone					Xbox				
Windows	**1**	**9**	**18**	**27**	**28**	**1**	**9**	**18**	**27**	**35**
Requests/Sec.	0.29	2.36	3.24	DNF	DNF	0.45	4.08	4.77	DNF	DNF
CPU Utilization %	8.2	55.93	100	DNF	DNF	7.13	57.23	100	DNF	DNF
Server Latency (ms)	681.96	762.31	4383.47	DNF	DNF	303.8	449.63	3473.5	DNF	DNF
Midori	**1**	**9**	**18**	**27**	**28**	**1**	**9**	**18**	**27**	**35**
Requests/Sec.	0.31	2.54	5.07	7.32	7.48	0.49	4.39	8.68	12.8	16.04
CPU Utilization %	2.98	24.58	49.51	83.72	86.69	3.99	23.72	44.11	65.78	93.72
Server Latency (ms)	193.53	213.68	234.28	606.85	735.64	5.82	32.24	51.49	94.29	297.46

Latency under load (lower is better)

Windows-based server does not finish



* Windows Speech Server cannot complete the benchmark at this concurrency level.

Result: SPECWeb05

Number of conforming connections (higher is better)

Workload	Windows	Midori	Ratio
Banking	34,800	69,000	198%
eCommerce	70,600	82,500	117%
Support (4 core)	27,300	47,000	172%
Weighted Average	**49,461**	**78,442**	**159%**

System

2x Quad Core CPU, 48GB RAM, 4x 512GB SSD, 2x 10Gb NIC + 2x 1Gb NIC
Logging and HTTPS enabled for both Windows and Midori runs per benchmark rules.
HyperThreading enabled.

Windows Settings

Same as official Windows submissions on SPEC.org, except for custom tunings that improves Windows performance such as affinity.
IPSEC and Firewall off

http://midori/Wiki Pages/Comparison of Midori and Windows SpecWeb05 Performance.aspx

Result: 10Gb Networking Performance

Bidirectional workload (both transmit and receive simultaneously)

- Windows: more than two CPU cores to saturate 10Gb wire
- Midori: less than a single CPU core to saturate 10Gb wire

System

Dell T5500, 2.4GHz CPU, Intel 82599 10Gb NIC

Result: SPECWeb05

Number of conforming connections (higher is better)

Workload	Windows	Midori	Ratio
Banking	34,800	69,000	198%
eCommerce	70,600	82,500	117%
Support (4 core)	27,300	47,000	172%
Weighted Average	**49,461**	**78,442**	**159%**

System

2x Quad Core CPU, 48GB RAM, 4x 512GB SSD, 2x 10Gb NIC + 2x 1Gb NIC
Logging and HTTPS enabled for both Windows and Midori runs per benchmark rules.
HyperThreading enabled.

Windows Settings

Same as official Windows submissions on SPEC.org, except for custom tunings that improves Windows performance such as affinity.
IPSEC and Firewall off

http://midori/Wiki Pages/Comparison of Midori and Windows SpecWeb05 Performance.aspx

Result: 10Gb Networking Performance

Bidirectional workload (both transmit and receive simultaneously)

- Windows: more than two CPU cores to saturate 10Gb wire
- Midori: less than a single CPU core to saturate 10Gb wire

System

Dell T5500, 2.4GHz CPU, Intel 82599 10Gb NIC

Result: SPECWeb05

Number of conforming connections (higher is better)

Workload	Windows	Midori	Ratio
Banking	34,800	69,000	198%
eCommerce	70,600	82,500	117%
Support (4 core)	27,300	47,000	172%
Weighted Average	**49,461**	**78,442**	**159%**

System

2x Quad Core CPU, 48GB RAM, 4x 512GB SSD, 2x 10Gb NIC + 2x 1Gb NIC
Logging and HTTPS enabled for both Windows and Midori runs per benchmark rules.
HyperThreading enabled.

Windows Settings

Same as official Windows submissions on SPEC.org, except for custom tunings that improves Windows performance such as affinity.
IPSEC and Firewall off

http://midori/Wiki Pages/Comparison of Midori and Windows SpecWeb05 Performance.aspx

http://midori/

We are hiring